¿Qué es REST? REST, REpresentational State Transfer

Es un tipo de arquitectura de desarrollo web que se apoya totalmente en el estándar HTTP.

REST nos permite crear servicios y aplicaciones que pueden ser usadas por cualquier dispositivo o cliente que entienda HTTP, por lo que es más simple y convencional que otras alternativas que se han usado en los últimos diez años como SOAP y XML-RPC.

REST se definió en el 2000 por Roy Fielding, coautor principal también de la especificación HTTP. Es un framework para construir aplicaciones web respetando HTTP.

REST es el tipo de arquitectura más natural y estándar para crear APIs para servicios orientados a Internet.

Existen tres niveles de calidad a la hora de aplicar REST en el desarrollo de una aplicación web y más concretamente una API que se recogen en un modelo llamado Richardson Maturity Model en honor al tipo que lo estableció, Leonard Richardson padre de la arquitectura orientada a recursos.

Estos niveles son:

1. Uso correcto de URIs
2. Uso correcto de HTTP.
3. Implementar Hypermedia.

Además de estas tres reglas, nunca se debe guardar estado en el servidor, toda la información que se requiere para mostrar la información que se solicita debe estar en la consulta por parte del cliente.

Al no guardar estado, no es necesario preocuparnos de temas como el almacenamiento de variables de sesión e incluso. Podemos jugar con distintas tecnologías para servir determinadas partes o recursos de una misma API.

Nivel 1: Uso correcto de URIs

Cuando desarrollamos una web o una aplicación web, las URLs nos permiten acceder a cada uno de las páginas, secciones o documentos del sitio web.
Cada página, información en una sección, archivo, cuando hablamos de REST, los nombramos como recursos.
El recurso por lo tanto es la información a la que queremos acceder o que queremos modificar o borrar, independientemente de su formato.

Las URL, Uniform Resource Locator , son un tipo de URI, Uniform Resource Identifier, que además de permitir identificar de forma única el recurso, nos permite localizarlo para poder acceder a él o compartir su ubicación.

Una URL se estructura de la siguiente forma:
{protocolo}://{dominio o hostname}[:puerto (opcional)]/{ruta del recurso}?{consulta de filtrado}

Reglas básicas para ponerle nombre a la URI de un recurso:

1. Los nombres de URI no deben implicar una acción, por lo tanto debe evitarse usar verbos en ellos.

2. Los nombres de URI deben ser únicas, no debemos tener más de una URI para identificar un mismo recurso.

Por ejemplo, la URI /facturas/234/editar sería incorrecta ya que tenemos el verbo editar en la misma.
Para el recurso factura con el identificador 234, la siguiente URI sería la correcta, independientemente de que vayamos a editarla, borrarla, consultarla o leer sólo uno de sus conceptos: /facturas/234

3. Deben ser independiente de formato.

Por ejemplo, la URI /facturas/234.pdf no sería una URI correcta, ya que estamos indicando la extensión pdf en la misma.
Para el recurso factura con el identificador 234, la siguiente URI sería la correcta, independientemente de que vayamos a consultarla en formato pdf, epub, txt, xml o json: /facturas/234

4. Deben mantener una jerarquía lógica.

Por ejemplo, la URI /facturas/234/cliente/007 no sería una URI correcta, ya que no sigue una jerarquía lógica.

5. Los filtrados de información de un recurso no se hacen en la URI.

Para el recurso factura con el identificador 234 del cliente 007, la siguiente URI sería la correcta: /clientes/007/facturas/234

Para filtrar, ordenar, paginar o buscar información en un recurso, debemos hacer una consulta sobre la URI, utilizando parámetros HTTP en lugar de incluirlos en la misma.
Por ejemplo, la URI /facturas/orden/desc/fecha-desde/2007/pagina/2 sería incorrecta ya que el recurso de listado de facturas sería el mismo pero utilizaríamos una URI distinta para filtrarlo, ordenarlo o paginarlo.

La URI correcta en este caso sería:
/facturas?fecha-desde=2007&orden=DESC&pagina=2

Nivel 2: HTTP

Para desarrollar APIs REST los aspectos claves que hay que dominar y tener claros:

- Métodos HTTP
- Códigos de estado
- Aceptación de tipos de contenido

Métodos.

Para manipular los recursos, HTTP nos dota de los siguientes métodos con los cuales debemos operar:

GET: *Para consultar y leer recursos*
Por ejemplo para un recurso de facturas.
GET /facturas Nos permite acceder al listado de facturas
GET /facturas/123 Nos permite acceder al detalle de una factura

POST: *Para crear recursos*
POST /facturas Nos permite crear una factura nueva

PUT: *Para editar recursos*
PUT /facturas/123 Nos permite editar la factura, sustituyendo la totalidad de la información anterior por la nueva.

DELETE: *Para eliminar recursos.*
DELETE /facturas/123 Nos permite eliminar la factura

PATCH: *Para editar partes concretas de un recurso.*
PATCH /facturas/123 Nos permite modificar cierta información de la factura, como el número o la fecha de la misma.

Códigos de estado.

Cuando realizamos una operación, es vital saber si dicha operación se ha realizado con éxito o en caso contrario, por qué ha fallado.

Un error común sería por ejemplo:

```
Petición
========
PUT /facturas/123

Respuesta
=========
Status Code 200
Content:
{
  success: false,
  code:    734,
  error:   "datos insuficientes"
}
```

En este ejemplo se devuelve un código de estado 200, que significa que la petición se ha realizado correctamente, sin embargo, estamos devolviendo en el cuerpo de la respuesta un error y no el recurso solicitado en la URL.
Este es un error común que tiene varios inconvenientes:
No es REST ni es estándar.
HTTP tiene un abanico muy amplio que cubre todas las posibles indicaciones que vamos a tener que añadir en nuestras respuestas cuando las operaciones han ido bien o mal.
Es imperativo conocerlos y saber cuándo utilizarlos.

El siguiente ejemplo sería correcto de la siguiente forma:

```
Petición
========
PUT /facturas/123

Respuesta
=========
Status Code 400
Content:
{
```

```
  message: "se debe especificar un id de cliente para la factura"
}
```

Tipos y formatos de contenido.

No es correcto indicar el tipo de formato de un recurso al cual queremos acceder o manipular en una URL.
HTTP nos permite especificar en qué formato queremos recibir el recurso, pudiendo indicar varios en orden de preferencia, para ello utilizamos el header Accept.
Nuestra API devolverá el recurso en el primer formato disponible y, de no poder mostrar el recurso en ninguno de los formatos indicados por el cliente mediante el header Accept, devolverá el código de estado HTTP 406.
En la respuesta, se devolverá el header Content-Type, para que el cliente sepa qué formato se devuelve, por ejemplo:

```
Petición
========
GET /facturas/123
Accept: application/epub+zip , application/pdf, application/json

Respuesta
=========
Status Code 200
Content-Type: application/pdf
```

En este caso, el cliente solicita la factura en epub comprimido con ZIP y de no tenerlo, en pdf o json por orden de preferencia. El servidor le devuelve finalmente la factura en pdf.

Nivel 3: Hypermedia.

A pesar de lo que nos pueda inducir a pensar el término retrofuturista Hypermedia, el concepto y la finalidad que busca describir es bastante sencillo: conectar mediante vínculos las aplicaciones clientes con las APIs, permitiendo a dichos clientes despreocuparse por conocer de antemano del cómo acceder a los recursos.
Con Hypermedia básicamente añadimos información extra al recurso sobre su conexión a otros recursos relacionados con él.
Aquí tenemos un ejemplo:

```
<pedido>
 <id>666</id>
 <estado>Procesado</estado>
 <links>
   <link rel="factura">
       http://example.com/api/pedido/666/factura
```

```
   </link>
 </links>
</pedido>
```

En este ejemplo vemos cómo indicar en un xml que representa un pedido, el enlace al recurso de la factura relacionada con el mismo.
Sin embargo, necesitamos que el cliente que accede a nuestra API entienda que esa información no es propia del recurso, sino que es información añadida que puede utilizar para enlazar el pedido con la factura.
Para ello conseguir esto, debemos utilizar las cabeceras Accept y Content-Type, para que tanto el cliente como la API, sepan que están hablando hypermedia.
Por ejemplo:

```
Petición
========
GET /pedido/666
Accept: application/nuestra_api+xml, text/xml

Respuesta
=========
Status Code: 200
Content-Type: application/nuestra_api+xml
Content:

<pedido>
 <id>666</id>
 <estado>Procesado</estado>
 <links>
   <link rel="factura">
        http://example.com/api/pedido/666/factura
   </link>
 </links>
</pedido>
```

Como vemos, el cliente solicita el formato application/nuestra_api+xml de forma preferente al formato text/xml. De esta forma, le indica al servicio web, que entiende su formato hypermedia y puede aprovecharlo.
El servicio web por lo tanto, como implementa hypermedia, le devuelve la información de recurso y la información de hypermedia que puede utilizar el cliente.
Hypermedia es útil por ejemplo para que el cliente no tenga que conocer las URLs de los recursos, evitando tener que hacer mantenimientos en cada uno de los mismos si en un futuro dichas URLs cambian (no debería ocurrir). También es útil para automatizar procesos entre APIs sin que haya interacción humana.